

Certified Connection Tableaux Proofs for HOL Light and TPTP

Cezary Kaliszyk

University of Innsbruck
cezary.kaliszyk@uibk.ac.at

Josef Urban

Radboud University Nijmegen
josef.urban@gmail.com

Jiří Vyskočil

Czech Technical University in Prague
jiri.vyskocil@gmail.com

Abstract

In the recent years, the Metis prover based on ordered paramodulation and model elimination has replaced the earlier built-in methods for general-purpose proof automation in HOL 4 and Isabelle/HOL. In the annual CASC competition, the leanCoP system based on connection tableaux has however performed better than Metis. In this paper we show how the leanCoP's core algorithm can be implemented inside HOL Light. leanCoP's flagship feature, namely its minimalistic core, results in a very simple proof system. This plays a crucial role in extending the MESON proof reconstruction mechanism to connection tableaux proofs, providing an implementation of leanCoP that certifies its proofs. We discuss the differences between our direct implementation using an explicit Prolog stack, to the continuation passing implementation of MESON present in HOL Light and compare their performance on all core HOL Light goals. The resulting prover can be also used as a general purpose TPTP prover. We compare its performance against the resolution based Metis on TPTP and other interesting datasets.

Categories and Subject Descriptors CR-number [subcategory]: third-level

Keywords keyword1, keyword2

1. Introduction and Related Work

The leanCoP [19] automated theorem prover (ATP) has an unusually good ratio of performance to its implementation size. While its core algorithm fits on some twenty lines of Prolog, starting with CASC-21 [27] it has regularly beaten Otter [17] and Metis [7] in the FOF division of the CASC ATP competitions. In 2014, leanCoP solved 158 FOF problems in CASC-J7,¹ while Prover9 solved 95 problems. On the large-theory (chainy) division of the MPTP Challenge benchmark², leanCoP's goal-directed calculus beats also SPASS 2.2 [31], and its further AI-style strengthening by integrating into leanCoP's simple core learning-based guidance trained on such larger ITP corpora is an interesting possibility [30].

¹<http://www.cs.miami.edu/~tptp/CASC/J7/WWWFiles/DivisionSummary1.html>

²<http://www.cs.miami.edu/~tptp/MPTPChallenge/>

Compact ATP calculi such as leanTAP [2] and MESON [16] have been used for some time in Isabelle [20, 21] and HOLs [4] as general first-order automation tactics for discharging goals that are already simple enough. With the arrival of large-theory “hammer” linkups [8, 10, 13, 22] between ITPs, state-of-the-art ATPs such as Vampire [12] and E [24], and premise selection methods [14], such tactics also became used as a relatively cheap method for reconstructing the (minimized) proofs found by the stronger ATPs. In particular, Hurd's Metis has been adopted as the main proof reconstruction tool used by Isabelle's Sledgehammer linkup [3, 23], while Harrison's version of MESON could reconstruct in 1 second about 80% of the minimized proofs found by E in the first experiments with the HOLyHammer linkup [9].

Since HOL Light already contains a lot of the necessary infrastructure for Prolog-style proof search and its reconstruction, integrating leanCoP into HOL Light in a similar way as MESON should not be too costly, while it could lead to interesting strengthening of HOL Light's first-order automation and proof reconstruction methods. In this paper we describe how this was done, resulting in an OCaml implementation of leanCoP and a general leanCoP first-order tactic in HOL Light. We compare their performance with MESON, Metis and the Prolog version of leanCoP in several scenarios, showing quite significant improvements over MESON and Metis.

2. leanCoP and Its Calculus

leanCoP is an automated theorem prover for classical first-order logic based on a compact Prolog implementation of the clausal connection (tableaux) calculus [15, 19] with several simple strategies that significantly reduce the search space on many problems. In contrast to saturation-based calculi used in most of the state-of-the-art ATPs (E, Vampire, etc.), connection calculi implement goal-oriented proof search. Their main inference step connects a literal on the current path to a new literal with the same predicate symbol but different polarity. The formal definition (derived from Otten [18]) of the particular *connection calculus* relevant in leanCoP is as follows:

DEFINITION 1. [*Connection calculus*] The axiom and rules of the *connection calculus* are given in Figure 1. The words of the calculus are tuples “ $C, M, Path$ ” where the clause C is the *open subgoal*, M is a set of clauses in disjunctive normal form (DNF) transformed from $axioms \wedge conjecture$ with added nullary predicate \sharp^3 to all positive clauses⁴, and the *active path* $Path$ is a subset of a path through M . In the rules of the calculus C, C' and C'' are clauses, σ is a term substitution, and L_1, L_2 is a *connection* with

³We suppose that \sharp is a new predicate that does not occur anywhere in *axioms* and *conjecture*

⁴Thus by default all positive clauses are used as possible start clauses.

$\sigma(L_1) = \sigma(\overline{L_2})$. The rules of the calculus are applied in an analytic (i.e. bottom-up) way. The term substitution σ is applied to the whole derivation.

The connection calculus is correct and complete [15] in the following sense: A first-order formula M in clausal form is valid iff there is a connection proof for “ $\neg\#, M, \{\}$ ”, i.e., a derivation for “ $\neg\#, M, \{\}$ ” in the connection calculus so that all leaves are axioms. The Prolog predicate `prove/5` implements the axiom, the reduction and the extension rule of the basic connection calculus in `leanCoP`:

```

1 %
2 prove([Lit|Cla],Path,PathLim,Lem,Set) :-
3     %
4     (-NegLit=Lit;~Lit=NegLit) ->
5     (
6         %
7         member(NegL,Path),
8         unify_with_occurs_check(NegL,NegLit)
9     );
10    lit(NegLit,NegL,Cla1,Grnd1),
11    unify_with_occurs_check(NegL,NegLit),
12    %
13    %
14    prove(Cla1,[Lit|Path],PathLim,Lem,Set)
15 ),
16 %
17 prove(Cla,Path,PathLim,Lem,Set).
18 prove([],_,_,_,_).
```

The tuple “ $C, M, Path$ ” in connection calculus is here represented as follows:

- C representing the open subgoal is a Prolog list `Cla`;
- the active path $Path$ is a Prolog list `Path`;
- M is written into Prolog’s database before the actual proof search starts in a such way that for every clause $C \in M$ and for every literal $C \in M$ the fact `lit(Indexing_L,L,C1,Grnd)` is stored, where $C1=C \setminus \{L\}$ and `Grnd` is `g` if C is ground, otherwise `Grnd` is `n`. `Indexing_L` is same as L modulo all its variables which are fresh (there is no twice or more occurrences in `Indexing_L`) everywhere in `Indexing_L` and it is used for fast finding the right fact in database without affecting the logically correct L by standard Prolog unification without occurs check.
- Atoms are represented by Prolog atoms, negation by “ \neg ”.
- The substitution σ is stored implicitly by Prolog.

`PathLim` is the current limit used for iterative deepening, `Lem` is the list of usable (previously derived) lemmas, `Set` a list of options, and `Proof` is the resulting proof. This predicate succeeds (using iterative deepening) iff there is a connection proof for the tuple represented by `Cla`, the DNF representation of the problem stored in Prolog’s database using the `lit` predicate, and a `Path` with $|Path| < PathLim$ where `PathLim` is the maximum size of the active `Path`. The predicate works as follows:

Line 18 implements the axiom, line 4 calculates the complement of the first literal `Lit` in `Cla`, which is used as the principal literal for the next reduction or extension step. The reduction rule is implemented in lines 7, 8 and 17. At line 7 and 8 it is checked whether the active path `Path` contains a literal `NegL` that unifies with the complement `NegLit` of the principal literal `Lit`. In this case the alternative lines after the semicolon are skipped and the proof search for the premise of the reduction rule is invoked in line 17. The extension rule is implemented in lines 10, 11, 14 and 17. Lines 10 and 11 are used to find a clause that contains the complement `NegLit` of the principal literal `Lit`. `Cla1` is the remaining set of literals of the selected clause and the new open

axiom: $\frac{}{\{\}, M, Path}$

reduction rule: $\frac{C, M, Path \cup \{L_2\}}{C \cup \{L_1\}, M, Path \cup \{L_2\}}$
where there exists a unification substitution σ such that $\sigma(L_1) = \sigma(\overline{L_2})$

extension rule: $\frac{C' \setminus \{L_2\}, M, Path \cup \{L_1\} \quad C, M, Path}{C \cup \{L_1\}, M, Path}$
where C' is a fresh copy of some $C'' \in M$ such that $L_2 \in C'$ and $\sigma(L_1) = \sigma(\overline{L_2})$ where σ is unification substitution.

Note that the σ used in the *reduction* and *extension* rules must be applied on all literals in all derivations except the literals in the set M because these literals are not affected by any substitution σ .

Figure 1. The basic clause connection calculus used in `leanCoP`.

subgoal of the left premise. The proof search for the left premise of the extension rule, in which the active path `Path` is extended by the principal literal `Lit`, is invoked in line 14, and if successful, we again continue on line 17.

Compared to standard tableaux or sequent calculi, connection calculi are not confluent⁵. To achieve completeness, an extensive use of backtracking is required. `leanCoP` uses two simple incomplete strategies (namely options `scut` and `cut`) for restricting backtracking that significantly reduces the search space [18] without affecting the ability to find proofs in most tested cases (see Section 4).

Another major problem in connection calculi is the integration of equality. The paramodulation method that is widely used in saturation-based ATPs is not complete for goal-oriented approach of connection calculi. Therefore equality in `leanCoP` and similar ATPs is usually managed by adding the axioms of equality (reflexivity, symmetry, transitivity and substitutivity).

To obtain the clausal form, `leanCoP` uses its own implementation of clausifier introducing definitions (the `def` option), which seems to perform better with the `leanCoP`’s core prover than other standard clausifiers (TPTP2X using the option `-t clausify:tptp`, FLOTTER and E) or direct transformation into clausal form (`nodef` option in `leanCoP`) [18]. In the following subsections, we summarize several further methods used by `leanCoP` that improve its performance.

2.1 Iterative deepening

Prolog uses a simple incomplete depth-first search strategy to explore the search space. This kind of incompleteness would result in a calculus that hardly proves any formula. In order to obtain a complete proof search in the connection calculus, iterative deepening on the proof depth, i.e. the size of the active path, is performed. It is achieved by inserting the following lines into the code:

```

(12) ( Grnd1=g -> true ; length(Path,K),
      K<PathLim -> true ;
(13) \+ pathlim -> assert(pathlim), fail ),
```

and the whole prover runs in the following iterative sense starting from `PathLimit=1`:

```

prove(PathLim,Set) :-
    retract(pathlim) ->
    PathLim1 is PathLim+1,
    prove(PathLim1,Set).
```

When the extension rule is applied and the new clause is not ground, i.e. it does not contain any variable, it is checked whether the size K of the active path exceeds the current path limit `PathLim`

⁵ Bad choice of *connection* might end up in dead end

(line 12). In this case the dynamic predicate `pathlim/0` is written into the Prolog's database (line 13) indicating the need to increase the path limit if the proof search with the current path limit fails. If the proof search fails and the predicate `pathlim` can be found in the database, then `PathLim` is increased by one and the proof search starts again.

2.2 Regularity Condition Optimization

DEFINITION 2. A connection proof is *regular* iff no literal occurs more than once in the active path.

Since the active path corresponds to the set of literals in a branch in the connection tableau representation, a connection tableau proof is regular if in the current branch no literal occurs more than once. The regularity condition is integrated into the connection calculus in Figure 1 by imposing the following restriction on the reduction and extension rule: $\forall L' \in C \cup \{L_1\} : \sigma(L') \notin \sigma(Path)$

LEMMA 2.1. A formula M in clausal form described above is valid iff there is a regular connection proof for " $\neg\#, M, \{\}$ "

Regularity is correct, since it only imposes a restriction on the applicability of the reduction and extension rules. The completeness proof can be found in [15, 19].

The regularity condition must be checked whenever the reduction, extension or lemma rule is applied. The substitution σ is not modified, i.e. the regularity condition is satisfied if the open subgoal does not contain a literal that is syntactically identical with a literal in the active path. This is implemented by inserting the following line into the code:

```
(3) \+ (member(LitC,[Lit|Cla]),
      member(LitP,Path),
      LitC==LitP),
```

The Prolog predicate `\+ Goal` succeeds only if *Goal* cannot be proven. In line 3 the corresponding *Goal* succeeds if the open subgoal `[Lit|Cla]` contains a literal `LitC` that is syntactically identical (built-in predicate `==/2` in Prolog) with a literal `LitP` in the active path `Path`. The built-in predicate `member/2` is used to enumerate all elements of a list.

2.3 Lemmata optimization

The set of lemmata is represented by the list `Lem`. The lemma rule is implemented by inserting the following lines:

```
(5) ( member(LitL,Lem), Lit==LitL
(6) ;
```

In order to apply the lemma rule, the substitution σ is not modified, i.e. the lemma rule is only applied if the list of lemmata `Lem` contains a literal `LitL` that is syntactically identical with the literal `Lit`. Furthermore, the literal `Lit` is added to the list `Lem` of lemmata in the (left) premise of the reduction and extension rule by adapting the following line:

```
(15) prove(Cla,Path,PathLim,[Lit|Lem],Set).
```

In the resulting implementation, the lemma rule is applied before the reduction and extension rules.

2.4 Restricted backtracking

In Prolog the cut (!) is used to cut off alternative solutions when Prolog tries to prove a goal. The Prolog cut is a built-in predicate, which succeeds immediately when first encountered as a goal. Any attempt to re-satisfy the cut fails for the parent goal, i.e. other alternative choices are discarded that have been made from the point when the parent goal was invoked. Consequently, *restricted backtracking* is achieved by inserting a Prolog cut after the lemma,

reduction, or extension rule is applied. It is implemented by inserting the following line into the code:

```
(16) ( member(cut,Set) -> ! ; true ),
```

Restricted backtracking is switched on if the list `Set` contains the option `cut`.

The *restricted start step* is used if the list `Set` includes the option `scut`. In this case only the first matching clause to starting $\neg\#$ literal is used.

Restricted backtracking and restricted start step lead to an incomplete proof search. In order to regain completeness, these strategies can be switched off when the search reaches a certain path limit. If the list `Set` contains the option `comp(Limit)`, where *Limit* is a natural number, the proof search is stopped and started again without using incomplete search strategies.

3. OCaml Implementation

In this section, we first discuss our implementation⁶ of leanCoP in OCaml and its integration in HOL Light: the transformation of the higher-order goal to first order and the proof reconstruction. After that we compare our implementation to Harrison's implementation of MESON.

3.1 leanCoP in OCaml

Otten's implementation of leanCoP uses the Prolog search, backtracking, and indexing mechanisms to implement the connection tableaux proof search. This is a variation of the general idea of using the "Prolog technology theorem prover" (PTTP) proposed by Stickel [25], in which connection tableaux takes a number of advantages from its similarity to Prolog.

In order to implement an equivalent program in a functional programming language, one needs to use either an explicit stack for keeping track of the current proof state (including the trail of variable bindings), or the continuation passing style. We choose to do the former, namely we add an explicit `todo` (stack), `subst` (trail) and `off` (offset in the trail) arguments to the main `prove` function. The stack keeps a list of tuples that are given as arguments to the recursive invocations of `prove`, whose full OCaml declaration (taking the open subgoal as its last argument) then looks as follows:

```
let rec lprove off subst path limit lemmas todo = function
[] -> begin ... end
| ((lit1 :: rest_clause) as clause) -> ... ;;
```

The function performs the proof search to the given depth, and if a proof has not been found, it returns the unit. It takes special attention to traverse the tree in the same order as the Prolog version. In particular, when the global option "cut" (restricting backtracking) is off, it performs all the backtrackings explicitly, while if "cut" is on, the parts of backtracking avoided in Prolog are also omitted. When a proof is found, the exception 'Solved' is raised: no further search is performed and the function exits with this exception.

The OCaml version and the Prolog version (simplified and with symbols renamed for clarity of comparison) are displayed together in Fig. 2. The algorithm proceeds as follows:

1. If nonempty, decompose the current open subgoal into the first literal `lit` and the rest `cla`.
2. Check for intersection between the current open subgoal and `path`.
3. Compute the negation of `lit`.

⁶ Available online at <http://cl-informatik.uibk.ac.at/users/cek/cpp15>

<pre> 1 let rec prove path lim lem stack = function (lit :: cla) -> 2 if not ((exists (fun litp -> exists (substeq litp) path)) (lit :: cla)) then (3 let neglit = negate lit in 4 if not (exists (substeq lit) lem && (prove path lim lem stack cla; cut)) then (5 if not (fold_left (fun sf plit -> if sf then true else try (unify_lit neglit plit; prove path lim (lit :: lem) stack cla; cut) with Unify -> sf) false path) then (6 let iter_fun (lit2, cla2, ground) = if lim > 0 ground then try let clal = unify_rename (snd lit) (lit2, cla2) in prove (lit :: path) (lim - 1) lem ((if cut then lim else -1), path, lim, lit :: lem, cla) :: stack) clal with Unify -> () in try iter_fun (try assoc neglit lits with Not_found -> []) with Cut n -> if n = lim then () else raise Cut n))) [] -> match stack with (ct, path, lim, lem, cla) :: t -> prove path lim lem t cla; if ct > 0 raise (Cut ct) 7 [] -> raise Solved;; </pre>	<pre> prove([Lit Cla],Path,PathLim,Lem,Set) :- \+ (member(LitC,[Lit Cla]), member(LitP,Path), LitC==LitP), (-NegLit=Lit;-Lit=NegLit) -> (member(LitL,Lem), Lit==LitL ; member(NegL,Path), unify_with_occurs_check(NegL,NegLit) ; lit(NegLit,NegL,ClaL,Grnd1), unify_with_occurs_check(NegL,NegLit), (Grnd1=g -> true ; length(Path,K), K<PathLim -> true ; \+ pathlim -> assert(pathlim), fail), prove(ClaL,[Lit Path],PathLim,Lem,Set)), (member(cut,Set) -> ! ; true), prove(Cla,Path,PathLim,[Lit Lem],Set). prove([],_,_,_,[]). </pre>
---	--

Figure 2. The simplified OCaml and Prolog code side by side. The explicit trail argument and the computation of the resulting proof have been omitted for clarity, and some symbols were renamed to better correspond to each other. White-space and order of clauses has been modified to exemplify corresponding parts of the two implementations. Function `substeq` checks equality under the current context of variable bindings. Note that the last-but-one line in the Prolog code was merged into each of the three cases in the OCaml code. See the function `lprove` in file `leanCoP.ml` on our web page for the actual (non-simplified) OCaml code.

4. Check if `lit` is among the lemmas `lem`, if so try to prove `cla`. If `cut` is set, no other options are tried.
5. For each literal on the path, if `neglit` unifies with it, try to prove `cla`. If the unification succeeded and `cut` is set, no other options are tried.
6. For each clause in the matrix, try to find a literal that unifies with `neglit`, and then try to prove the rest of the newly created subgoal and the rest of the current open subgoal. If the unification and the first proof succeeded and `cut` is set, no other options are tried.
7. When the current open subgoal is empty, the subproof is finished (the axiom rule).

In Otten’s implementation, the behaviour of the program with `cut` set is enabled by the use of the Prolog `cut (!)`. Implementing it in OCaml amounts to a different mechanism in each of the three cases. In point 4 in the enumeration above, given that a single lemma has been found, there is no need to check for other lemmas. Therefore a simple `List.exists` call is sufficient to emulate this behaviour in OCaml. No backtracking over other possible occurrences of the lemma is needed here, and it is not necessary to add in this case the literal again into the list of lemmas as is done in the Prolog code (last-but-one line).

In point 5, multiple literals on the path may unify with different substitutions. We therefore use a list fold mechanism which changes the value whenever the unification is successful and `cut` is set. In point 6, we need to change our behaviour in between two successive calls to `prove`. As the first call takes the arguments to the second call on the stack, we additionally add a cut marker on the stack and handle an exception that can be raised by the call on the stack.

Whenever the clause becomes empty, all the tuples in the stack list need to be processed. For each tuple, the first component is the cut marker: if it is set, the `Cut` exception is raised with a depth level. The exception is handled only at the appropriate level.

This directly corresponds to the semantics of the cut operator in Prolog [26].

What remains to be implemented is efficient equality checking and unification. Since we want to integrate our mechanism in HOL Light, we reuse the first order logic representation used in the implementation of HOL Light’s MESON procedure: the substitutions are implemented as association lists, and applications of substitutions are delayed until an actual equality check or a unification step.

3.2 leanCoP in HOL Light

In order to enable the OCaml version of leanCoP as a proof tactic and procedure in HOL Light, we first need to transform a HOL goal to a leanCoP problem and when a proof has been found we replay the proof in higher-order logic. In order to transform a problem in higher-order logic to first-order logic without equality, we mostly reuse the steps of the transformation already used by MESON, additionally ensuring that the conjecture is separated from the axioms to preserve leanCoP’s goal-directed approach. The transformation starts by assuming the duplicated copies of polymorphic theorems to match the existing assumptions. Next the goal $axioms \rightarrow conjecture$ is transformed to $(axioms \wedge \#) \rightarrow (conjecture \wedge \#)$ with the help of a special symbol, which we define in HOL as: $\# = \top$. Since the conjecture is refuted and the problem is converted to CNF, the only positive occurrence of $\#$ is present in a singleton clause, and the negative occurrences of $\#$ are present in every clause originating from the conjecture. The CNF clauses directly correspond to the DNF used by the Prolog version of leanCoP. Since no steps distinguish between the positivity of literals, the two can be used interchangeably in the proof procedure. We start the FOL algorithm by trying to prove $\neg\#$.

Since the final leanCoP proof may include references to lemmas, the reconstruction cannot be performed the same way as it is done in MESON. There, a tree structure is used for finished proofs. Each subgoal either closes the branch (the literal is a negation of a literal already present on the path) or is a branch extension with

a (possibly empty) list of subgoals. In leanCoP, each subgoal can refer to previous subgoals, so the order of the subgoals becomes important. We therefore flatten the tree to a list, which needs to be traversed in a linear order to reconstruct the proof.

We define a type of proof steps, one for each proof step in the calculus. Each application of a lemma step or path step constructs a proof step with an appropriate first-order term. For an application of a tableaux extension step we use Harrison’s contrapositive mechanism: we store the reference to the actual transformed HOL theorem whose conclusion is a disjunction together with the number of the disjunct that got resolved⁷.

```
type proof = Lem of fol_atom
           | Pat of fol_atom
           | Res of fol_atom * (int * thm);;
```

A list of such proof steps together with a final substitution and an initially empty list of already proved lemmas are the complete input to the proof reconstruction procedure. The reconstruction procedure always looks at the first step on the list. First, a HOL term is constructed from the FOL term with the final substitution applied. This step is straightforward, as it amounts to reversing the mapping of variables and constants applied to transform the HOL CNF to FOL CNF, with new names invented for new variables. Next, we analyze the kind of the step. If the step is a path step, the theorem $tm \vdash tm$ is returned, using the HOL ASSUME proof rule. If the step is a lemma step, the theorem whose conclusion is equal to tm is found on the list of lemmas and returned. Finally, if the proof step is an extension step, we first find the disjuncts of the HOL theorem in the proof step apart from the one that got matched. We then fold over this list, at every step calling the reconstruction function recursively with the remaining proof steps and the list of lemmas extended by each of the calls. The result of the fold is the list of theorems $[\vdash tm_1, \vdash tm_2, \dots, \vdash tm_n]$ which gets matched with the contrapositive theorem $\vdash tm_1 \wedge \dots \wedge tm_n \rightarrow tm_g$ using the HOL proof rule MATCH_MP to obtain the theorem $\vdash tm_g$. Finally, by matching this theorem to the term tm the theorem $\vdash tm$ is obtained.

As the reconstruction procedure traverses the list, it produces the theorem that corresponds to the first goal, namely $\dots \vdash \neg \sharp$. By unfolding the definition of \sharp , we obtain $\dots \vdash \perp$ which concludes the refutation proof.

3.3 Comparison to MESON

The simplified OCaml code of the core HOL Light’s MESON algorithm as described in [5] is as follows:

```
let rec mexpand rules ancestors g cont (env,n,k) =
  if n < 0 then failwith "Too_deep" else
  try tryfind (fun a -> cont (unify_literals
                             env (g,negate a),n,k))
    ancestors
  with Failure _ -> tryfind (fun rule ->
    let (asm,c), k = renamerule k rule in
    itlist (mexpand rules (g::ancestors)) asm cont
    (unify_literals env (g,c),n - length asm,k ))
    rules;;

let puremeson fm =
  let cls = simpcnf(specialize(pnf fm)) in
  let rules = itlist ((@) ** contrapositives) cls [] in
  deepen (fun n -> mexpand rules [] False (fun x -> x)
    (undefined,n,0); n)
  0;;
```

The toplevel puremeson function proceeds by turning the input formula into a clausal form, making contrapositives (rules)

⁷Contrary to the name, the HOL Light type fol_atom implements a literal: it is either positive or negative.

from the clauses, and then repeatedly calling the mexpand function with these rules using iterative deepening over the number of nodes permitted in the proof tree.

The mexpand function takes as its arguments the rules, an (initially empty) list of goal ancestors, the goal g to prove (initially False, which was also added to all-negative clauses when creating contrapositives), a continuation function cont for solving the rest of the subgoals (initially the identity), and a tuple consisting of the current trail env, the number n of additional nodes in the proof tree permitted, and a counter k for variable renaming.

If the allowed node count is not negative, mexpand first tries to unify the current goal with a negated ancestor, followed by calling the current continuation (trying to solve the remaining goals) with the extended trail. If all such unification/continuation attempts fail (i.e., they throw Failure), an extension step is tried with all rules. This means that the head of a (renamed) rule is unified with the goal g , the goal is appended to the ancestors and the mexpand is called (again using list folding with the subsequently modified trail and continuation) for all the assumptions of the rule, decreasing the allowed node count for the recursive calls.

The full HOL Light version of MESON additionally uses a smarter (divide-and-conquer) policy for the size limit, checks the goal for being already among the ancestors, caches continuations, and uses simple indexing. Below we enumerate some of the most important differences between the leanCoP algorithm and MESON and their implementations in HOL Light. Their practical effect is measured in Section 4.

- leanCoP computes and uses lemmas. The literals that correspond to closed branches are stored in a list. Each call to the main prove function additionally looks for the first literal in the list of lemmas. This can cost a linear number of equality checks if no parts of the proof are reused, but it saves computations if there are repetitions.
- Both algorithms use iterative deepening; however the depth and termination conditions are computed differently.
- MESON is implemented in the continuation-passing-style, so it can use as an additional optimization caching of the continuations. If any continuations are repeated (at the same depth level), the subproof is not retried. Otten’s leanCoP uses a direct Prolog implementation which cannot (without further tricks) do such repetition elimination. The implementation of leanCoP in OCaml behaves the same.
- leanCoP may use the cut after the lemma step, path step or successful branch closing in the extension step. Implementing this behaviour in OCaml exactly requires multiple Cut exceptions – one for each depth of the proof.
- The checking for repetitions is done in a coarser way in MESON than in leanCoP, allowing leanCoP to skip some work done by MESON.
- The search is started differently in leanCoP and in MESON. leanCoP starts with a conjecture clause, which likely contributes to its relatively good performance on larger problems.

4. Experimental Setup and Results

For the experiments we use HOL Light SVN version 199 (September 2014), Metis 2.3, and leanCoP 2.1. Unless noted otherwise, the systems are run on a 48-core server with AMD Opteron 6174 2.2 GHz CPUs, 320 GB RAM, and 0.5 MB L2 cache per CPU. Each problem is always assigned one CPU.

The systems are compared on several benchmarks, corresponding to different modes of use: goals coming from HOL Light itself,

Prover	Theorem (%)	Unique
mlleancop-cut-comp	759 (87.04)	2
mlleancop-nocut	759 (87.04)	2
plleancop-cut	752 (86.23)	0
plleancop-nc	751 (86.12)	0
metis-23	708 (81.19)	26
meson	683 (78.32)	4
any	832 (95.41)	

Table 1. Core HOL Light MESON calls without splitting (872 goals), 5sec per goal

the general set of problems from the TPTP library, and the large-theory problems extracted from Mizar [28]. The first set of problems is important, however since these problems come from HOL Light itself, they are likely naturally biased towards MESON. The Mizar problems come from the two MPTP-based benchmarks: the MPTP Challenge and MPTP2078 [1]. These are large-theory problems coming from a different ITP, hence they do not introduce the implicit bias as the HOL Light problems, while coming from a more relevant application domain than the general TPTP problems.

For HOL Light, we evaluate (with 5 second time limit) on two sets of problems. First, we look at 872 MESON-solved HOL Light goals that were made harder by removing splitting. In this scenario the tactic is applied to a subgoal of a proof, which is a bit similar to the Judgement-Day [3] evaluation used for Isabelle/Sledgehammer, where the goals are however restricted to the solvable ones. Table 1 shows the results. Second, we evaluate on the top-level goals (with their dependencies already minimized) that have been solved with the HOLyHammer system [11], i.e., by using the strongest available ATPs. This set is important because tactics such as MESON, Metis and now also leanCoP can be tried as a first cheap method for reconstructing the proofs found by the stronger ATPs. The results are shown in Table 2. In both cases, the OCaml implementation of leanCoP performs best, improving the MESON’s performance in the first case by about 11%, and improving on Metis on the second set of problems by about 45%.

Table 3 shows the results of the evaluation on all 7036 FOF problems coming from TPTP 6.0.0, using 10 second time limit. Here the difference to Metis is not so significant, probably because Metis implements ordered paramodulation, which is useful for many TPTP problems containing equality. The improvement over MESON is about 17%. Table 4 and Table 5 show the results on the small (heuristically minimized) and large MPTP Challenge problems. The best version of the OCaml implementation of leanCoP improves by 54% on Metis and by 90% on MESON on the small problems, and by 88% on Metis and 100% on MESON on the large problems. Here the goal directedness of leanCoP is probably the main factor.

Finally, to get a comparison also with the best ATPs on a larger ITP-oriented benchmark (using different hardware), we have done a 10s evaluation of several systems on the newer MPTP2078 benchmark (used in the 2012 CASC@Turing competition), see Table 6 and Table 7. The difference to Metis and MESON on the small problems is still quite significant (40% improvement over MESON), while on the large problems the goal-directedness again shows even more (about 90% improvement). While Vampire’s (version 2.6) SInE heuristic [6] helps a lot on the larger problems [29], the difference there between E (1.8) and our version of leanCoP is not so great as one could imagine given the several orders of magnitude difference in the size of their implementations.

Prover	Theorem (%)	Unique
mlleancop-cut-comp	1178 (75.70)	12
mlleancop-nocut	1162 (74.67)	0
meson	1110 (71.33)	39
plleancop-nc	1085 (69.73)	0
plleancop-cut	1084 (69.66)	0
metis-23	814 (52.31)	16
any	1260 (80.97)	

Table 2. HOL Light dependencies (1556 goals, 5sec)

Prover	Theorem (%)	Unique
mlleancop-cut-conj	1669 (23.72)	73
plleancop-cut-conj	1648 (23.42)	21
plleancop-cut	1622 (23.05)	34
mlleancop-cut	1571 (22.32)	9
metis-23	1562 (22.20)	261
meson	1430 (20.32)	28
plleancop-nocut	1358 (19.30)	25
mlleancop-nocut	1158 (16.45)	3
any	2433 (34.57)	

Table 3. TPTP (7036 goals with at least one conjecture, 10sec)

Prover	Theorem (%)	Unique
pllean-cut-conj	103 (40.87302)	2
pllean-cut	99 (39.28571)	8
mlleancop-cut-conj	91 (36.11111)	2
mlleancop-cut	79 (31.34921)	0
mlleancop-nocut	76 (30.15873)	0
pllean-nc	62 (24.60317)	1
metis-23	59 (23.41270)	3
meson-infer	48 (19.04762)	0
any	124 (49.20635)	

Table 4. Bushy (small) MPTP Challenge problems (252 in total), 10sec)

Prover	Theorem (%)	Unique
pllean-cut-conj	61 (24.20635)	5
mlleancop-cut-conj	60 (23.80952)	9
pllean-cut	57 (22.61905)	4
mlleancop-nocut	47 (18.65079)	0
mlleancop-cut	47 (18.65079)	0
metis-23	32 (12.69841)	3
meson-infer	30 (11.90476)	0
pllean-nc	26 (10.31746)	0
any	83 (32.93651)	

Table 5. Chainy (large) MPTP Challenge problems (252 in total), 10sec)

Prover	Theorem (%)
Vampire	1198 (57.65)
e18	1022 (49.18)
mlleancop-cut-conj	613 (29.49)
pllean-cut-conj	597 (28.72)
metis-23	564 (27.14)
mlleancop-cut	559 (26.90)
pllean-cut	544 (26.17)
pllean-comp7	539 (25.93)
mlleancop-nocut	521 (25.07)
pllean-nc	454 (21.84)
meson-infer	438 (21.07)
any	1277 (61.45)

Table 6. Bushy (small) MPTP2078 problems (2078 in total), 10sec)

Prover	Theorem (%)
Vampire	634 (30.51)
e18	317 (15.25)
mlleancop-cut-conj	243 (11.69)
pllean-cut-conj	196 (9.43)
pllean-cut	170 (8.18)
pllean-comp7	159 (7.65)
mlleancop-nocut	150 (7.21)
mlleancop-cut	146 (7.02)
meson-infer	145 (6.97)
metis-23	138 (6.64)
pllean-nc	126 (6.06)
any	693 (33.34)

Table 7. Chainy (large) MPTP2078 problems (2078 in total), 10sec)

5. Conclusion

We have implemented an OCaml version of the leanCoP compact connection prover, and the reconstruction of its proofs inside HOL Light. This proof-reconstruction functionality can be also used to certify in HOL Light an arbitrary TPTP proof produced by leanCoP, thus turning leanCoP into one of the few ATPs whose proofs enjoy LCF-style verification in one of the safest LCF-based systems. The performance of the OCaml version on the benchmarks is comparable to the Prolog version, while it always outperforms Metis and MESON, sometimes very significantly on the relevant ITP-related benchmarks.

We provide a HOL Light interface that is identical to the one offered by MESON, namely we provide two tactics and a rule. `LEANCOPTAC` and `ASM_LEANCOPTAC` are given a list of helper theorems, and then try to solve the given goal together (or the given goal and assumptions, respectively). The `LEANCOPTAC` rule, given a list of helper theorems acts as a conversion, i.e., given a term statement it tries to prove a theorem whose conclusion is identical to that of the term. The benchmarks show that these are likely the strongest single-step proof-reconstructing first-order tactics available today in any ITP system.

Acknowledgments

Supported by the Austrian Science Fund (FWF): P26201.

References

- [1] J. Alama, T. Heskes, D. Kühlwein, E. Tsivtsivadze, and J. Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2):191–213, 2014.
- [2] B. Beckert and J. Posegga. leanTAP: Lean tableau-based deduction. *J. Autom. Reasoning*, 15(3):339–358, 1995.
- [3] S. Böhme and T. Nipkow. Sledgehammer: Judgement Day. In J. Giesl and R. Hähnle, editors, *IJCAR*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010.
- [4] J. Harrison. Optimizing Proof Search in Model Elimination. In M. McRobbie and J. Slaney, editors, *Proceedings of the 13th International Conference on Automated Deduction*, number 1104 in *LNAI*, pages 313–327. Springer, 1996.
- [5] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
- [6] K. Hoder and A. Voronkov. Sine qua non for large theory reasoning. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 299–314. Springer, 2011.
- [7] J. Hurd. First-order proof tactics in higher-order logic theorem provers. In M. Archer, B. D. Vito, and C. Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, Sept. 2003.
- [8] C. Kaliszyk and J. Urban. MizAR 40 for Mizar 40. *CoRR*, abs/1310.2805, 2013.
- [9] C. Kaliszyk and J. Urban. PRoCH: Proof reconstruction for HOL Light. In M. P. Bonacina, editor, *CADE*, volume 7898 of *LNCS*, pages 267–274. Springer, 2013.
- [10] C. Kaliszyk and J. Urban. HOL(y)Hammer: Online ATP service for HOL Light. *Mathematics in Computer Science*, 2014. <http://dx.doi.org/10.1007/s11786-014-0182-0>.
- [11] C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014.
- [12] L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.
- [13] D. Kühlwein, J. C. Blanchette, C. Kaliszyk, and J. Urban. MaSh: Machine learning for Sledgehammer. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 35–50. Springer, 2013.
- [14] D. Kühlwein, T. van Laarhoven, E. Tsivtsivadze, J. Urban, and T. Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In B. Gramlich, D. Miller, and U. Sattler, editors, *IJCAR*, volume 7364 of *LNCS*, pages 378–392. Springer, 2012.
- [15] R. Letz and G. Stenz. Model elimination and connection tableau procedures. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 2015–2114. Elsevier and MIT Press, 2001.
- [16] D. W. Loveland. Mechanical theorem proving by model elimination. *J. the ACM*, 15(2):236–251, Apr. 1968.
- [17] W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *J. Autom. Reasoning*, 18(2):211–220, 1997.
- [18] J. Otten. Restricting backtracking in connection calculi. *AI Commun.*, 23(2-3):159–182, 2010.
- [19] J. Otten and W. Bibel. leanCoP: lean connection-based theorem proving. *J. Symb. Comput.*, 36(1-2):139–161, 2003.
- [20] L. C. Paulson. Automated reasoning and its applications. chapter Generic Automatic Proof Tools, pages 23–47. MIT Press, Cambridge, MA, USA, 1997.
- [21] L. C. Paulson. A generic tableau prover and its integration with Isabelle. *J. UCS*, 5(3):73–87, 1999.
- [22] L. C. Paulson and J. Blanchette. Three years of experience with Sledgehammer, a practical link between automated and interactive theorem provers. In *8th IWIL*, 2010. Invited talk.

- [23] L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *TPHOLs*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.
- [24] S. Schulz. System description: E 1.8. In K. L. McMillan, A. Middeldorp, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 735–743. Springer, 2013.
- [25] M. E. Stickel. A Prolog Technology Theorem Prover: Implementation by an extended Prolog compiler. *J. Autom. Reasoning*, 4(4):353–380, 1988.
- [26] T. Ströder, F. Emmes, P. Schneider-Kamp, J. Giesl, and C. Fuhs. A linear operational semantics for termination and complexity analysis of ISO Prolog. In G. Vidal, editor, *LOPSTR*, volume 7225 of *LNCS*, pages 237–252, 2012.
- [27] G. Sutcliffe. The CADE-21 automated theorem proving system competition. *AI Commun.*, 21(1):71–81, 2008.
- [28] J. Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning*, 37(1-2):21–43, 2006.
- [29] J. Urban, K. Hoder, and A. Voronkov. Evaluation of automated theorem proving on the Mizar Mathematical Library. In *ICMS*, pages 155–166, 2010.
- [30] J. Urban, J. Vyskočil, and P. Štěpánek. MaLeCoP: Machine learning connection prover. In K. Brünner and G. Metcalfe, editors, *TABLEAUX*, volume 6793 of *LNCS*, pages 263–277. Springer, 2011.
- [31] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topić. System description: SPASS version 1.0.0. In *Automated Deduction - CADE-16*, volume 1632 of *LNCS*, pages 378–382. Springer Berlin Heidelberg, 1999.